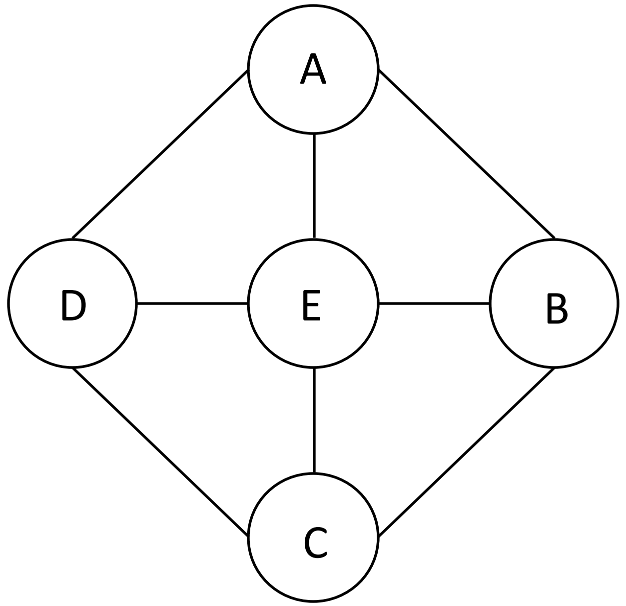
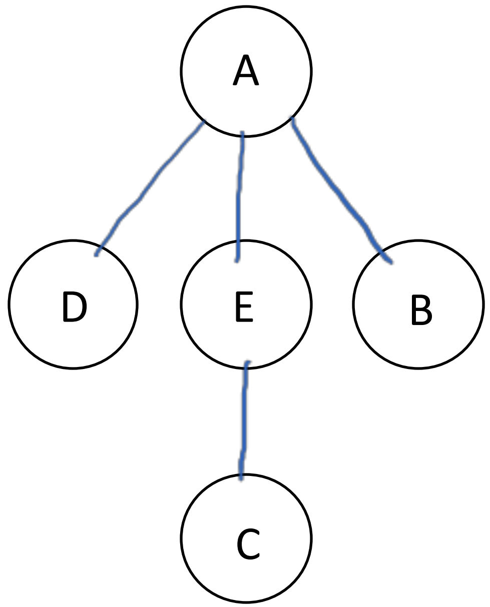
Spanning Trees

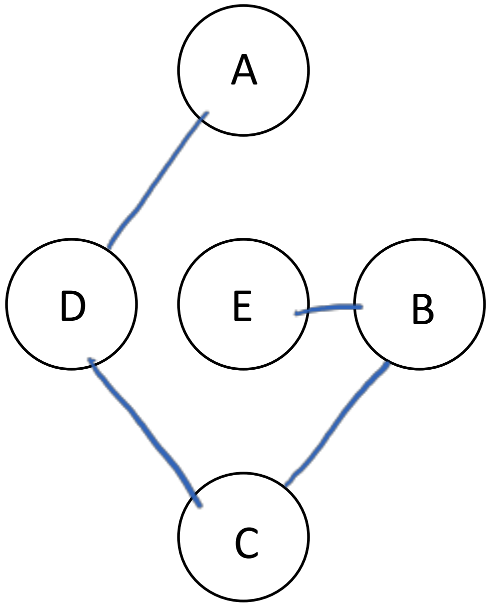
Definition

(★ Why is it important: It gives us the minimum set of edges we need in the graph, so that we could reach all the nodes in the graph.)

For an undirected graph G with V nodes: If T is a subgraph of G and T contains all nodes of G, is connected, and has exactly (V-1) edges, then T is a spanning tree of G.







Depth First Spanning Trees

DFT(v):

mark v as visited

for each unvisited successor u of v:

add v-u to spanning tree

DFT(u)

☆ Notice that the 2 traversal algorithms can also be used for finding the spanning tree. (Each time before we visit a node, we add that edge to the spanning tree. Since every node is visited exactly once, there will be V-1 edges created also)

Breadth First Spanning Trees

```
BFT(v):
        q = new Queue()
        mark v as visited
        q.enqueue(v)
        while (!q.isEmpty()):
                c = q.dequeue()
                for each unvisited successor u of c:
                        add edge c-u to spanning tree
                        mark u as visited
                        q.enqueue(u)
```

Minimum Spanning Trees

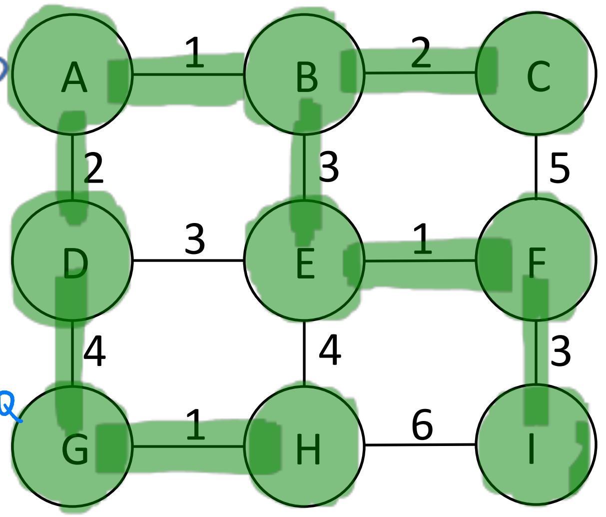
In a weighted graph:

Minimum spanning trees are the spanning trees with the lowest sum of edge weights.

Subset of the spanning tree

Prim’s Algorithm

Starting node



For weighted, connected, and undirected graphs.

① Insert all outgoing edges of the starting node into PQ

② Find the minimum edge in the PQ

③ If unvisited, mark visited and add the edge to tree

④ Repeat ①

Starting node

```
prim(v):
    pq = new PriorityQueue()
    mark v as visited
    pq.insert(v.outgoingEdges)
    while (!pq.isEmpty()):
        c = pq.removeMin()
        if (c.endNode is unvisited):
            mark c.endNode as visited
            add c to tree
            pq.insert(c.endNode.outgoingEdges to unvisited nodes)
```

$E \cdot \log E$

E iterations

$\log E$

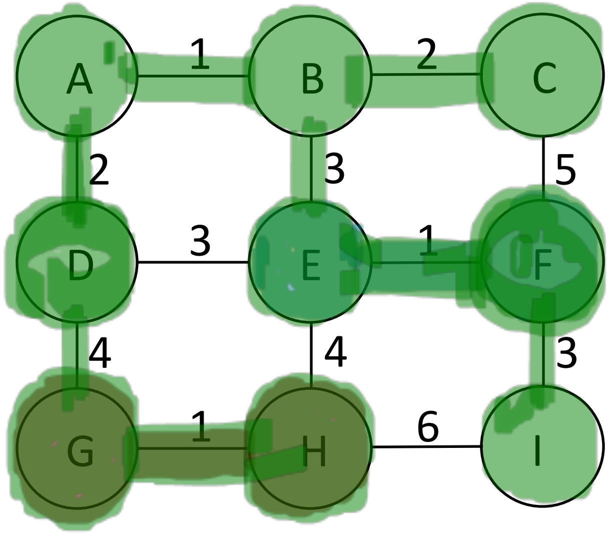
$4 \log E$

p.q.: ~~AB:1~~, ~~AD:2~~, ~~BC:2~~, ~~BE:3~~, ~~DE:3~~, ~~DG:4~~, ~~CF:5~~, ~~EF:1~~, ~~EH:4~~, ~~FI:3~~, ~~IH:6~~, ~~GH:1~~

$$\sum_{\text{edge weight}} = 17$$

$$\Rightarrow E \cdot \log E + E \cdot 2 \cdot \log E$$

Kruskal's Algorithm



For weighted, connected, and undirected graphs.

① First list all the edges in sorted order
② check if the start node / end node in different set
③ If yes, join them and add to the tree
④ Repeat ①

```
kruskal(edgeList):
    sort edgeList
    init nodeSets with singleton sets
    while (!edgeList.isEmpty()):
        c = edgeList.removeFirst()
        if (c.startNode and c.endNode in different nodeSets):
            add c to tree
            nodeSets.join(c.startNode, c.endNode)
```

sort edgeList → $E \cdot \log E$

init nodeSets → V

while → E iterations

if → $\log V$

join → $\log V$

edgeList: ~~AB:1~~, ~~EF:1~~, ~~GH:1~~, ~~BC:2~~, ~~AD:2~~, ~~BE:3~~, ~~DE:3~~, ~~FI:3~~, ~~DG:4~~, ~~EH:4~~, ~~CF:5~~, ~~HI:6~~

$\sum_{\text{edgeWeights}} = 17$

$\Rightarrow E \cdot \log E + V + E \cdot 2 \cdot \log V$

Complexities

V: # nodes in graph

E: # edges in graph

Prim: $O(\cancel{2} \cdot E \cdot \underline{\log E})$

$O(E \cdot \log V^2)$

$O(E \cdot \cancel{2} \cdot \underline{\log V})$

Kruskal:

$O(E \cdot \underline{\log E} + \cancel{V} + 2 \cdot E \cdot \log V)$

$O(E \cdot \log V^{\cancel{2}} + 2 \cdot E \cdot \log V)$

$O(E \cdot \log V)$